Model Classes for Dummies

Andrew Braun

April 7, 1998

Version 1.4

Contents

Chapter 1

Introduction

This book is written with the intent of being a stepping stone in the understanding and use of the accelerator modeling class libraries written by Leo Michelotti, Jim Holt and others. This book will not cover every aspect of each class structure and and its hierarchy, but will provide a useful starting point for understanding some of the more mundane uses of the model classes.

First of all, I am not a C++ programming guru. Nor am I a accelerator physicist. Rather, I am a user (as hopefully you will become) of the modeling code. These notes are a combination of questions and difficulties I and other have had when confronted with the seemingly massive amounts code and include files that comprise the class libraries. With sparse documentation, it is difficult to know where to start.

The libraries can be broken down into 4 major groupings. The first grouping consists of classes that allow the user to build beamlines out of various classes of beamline elements. The second major grouping of classes consists of an extensive library of Automatic Differentiation software for performing calculations of maps and Jets as well as support classes such as Vector, MatrixD, slist, dlist, and so on. The third major grouping of classes provide pre-compiled models of various accelerators such as the Tevatron in Collider mode, the Main Injector 8 GeV line and the Recycler. The last grouping of classes could be called support classes. These classes consist of code for managing servers, making Sybase queries, and providing an GUI interface via Tcl/Tk.

The code libraries can be used in various applications. Most of the programs the user will write will be programs that are run from a terminal in sort of a 'batch mode' fashion, printing their output to a file or to the screen.

The user can also use the code to make programs that provide a Graphical User Interface (GUI) where the user can select operations and queries to be performed on the beamline with the results displayed graphically. It is even possible to make a client/server application that listens for connections to a particular port and starts up a particular model based on messages the server receives on the port.

The user does not have to become proficient in all aspects of the class libraries to be able to perform basic tasks. The user can modify some of the skeleton programs in this book to suit their own needs and then add functionality as the task demands.

In the following chapters, some of the specifics of each group of classes will be presented along with some of the frequent uses of the classes to perform common tasks.

Chapter 2

Getting Started

This chapter will describe how to get and compile the libraries that make up the Accelerator model code written by Leo Michelotti and Jim Holt. I will assume that the user is on the CARTOON cluster and is fairly proficient in the Unix operating system.

2.1 Using pre-compiled libraries

The user of the software can use the libraries without really worrying about the guts of the software or even getting the source code. In the directory `/usr/local/ap/lattice_tools/model/fnal` are the include files and the libraries that the user can include and link to for their own programs. This alleviates the problem of retrieving and compiling the software in one's own directory. At this point the user is ready to start using the libraries to solve accelerator problems. A sample `Makefile` to compile a program called `twissCheck` utilizing these libraries is below:

```
# I am overriding the possible external definition
# of $FNALROOT so that the Makefile can stay
# essentially the same as the one used to
# link to locally compiled libraries.

FNALROOT = /usr/local/ap/lattice_tools/model/fnal
SYBASE = /usr/local/ap/sybase
```

```
# Define what directories will be searched for
# the include files.
include $FNALROOT/Make-config

INCLUDEDIR = -I$(BEAMLINE_INC) -I$(MXYZPTLK_INC) \
             -I$(MACHINE_INC)  -I$(FILTER_INC) \
             -I$(SYBASE_INC) -I$(TEV_INC) -I$(SWYD_INC) \
             -I$(BASIC_TOOLKIT_INC) -I$(PHYSICS_TOOLKIT_INC)

INCLUDE = -I. $(INCLUDEDIR) \
              -I/usr/local/TCL-TK/include \
      -I$(SYBASE)/include

# Define what directory the libraries should be found in.
MYLIBDIR = -L$(FNALROOT)/lib/sun

# Define what libraries we will link to.
MYLIBS =  -ltclgui -lsocket++ -lsybase -lfilter -lMachine \
          -lbeamline -lphysics_toolkit -lbasic_toolkit \

XLIBS = -L/usr/local/ap/X11R6/lib -L/usr/local/TCL-TK/lib \
        -ltcl -ltk \
        -lX11 -lICE -lSM -lsocket
SYSLIBS = -lcomplex -lm -lsocket

SYBASELIBS = -lct -lcs -lblk -lcomn -lsybtcl -lsybintl

# Define what will get built by default.
all:  twissCheck

C   = cc
C++ = CC
CFLAGS = -g
C++FLAGS = -g -sb -ptr$(FNALROOT) -ptv

.SUFFIXES: .o .C .c .cc

.C.o:
```

```
$(C++) $(C++FLAGS) $(INCLUDE) -c $*.C

.cc.o:
$(C++) $(C++FLAGS) $(INCLUDE) -c $*.cc

.c.o:
$(CC) $(CFLAGS) $(INCLUDE) -c $*.c

twissCheck: twissCheck.o
$(C++) $(C++FLAGS) -o twissCheck  twissCheck.o\
               $(MYLIBDIR) $(MYLIBS) $(SYSLIBS)

clean:
rm -f *.o
rm -rf .sb
rm -f *~*
rm -f twissCheck

depend:
makedepend -- $(DEPEND_INC) -- $(SOURCES)

# DO NOT DELETE THIS LINE -- make depend depends on it.
```

2.2 Retrieving the source code for code contributions

If the user is planning to contribute to the development of the software by extending the libraries, then it becomes necessary to get a copy of all of the software source code. To begin with, the software resides in a CVS (Control Version Software) repository on the cluster from which users and developers can checkout the latest copy of the software. The user will retrieve the software using the `cvs` command.

To retrieve the software from the `cvs` repository, you you will need to set a Unix environment variable to point to where the main `cvs` repository is.

```
setenv CVSROOT /home/room1/CVS
```

For users of `csh` or its variants this definition should be entered in their `.login` file or put in a file and executed each time before using the software. The variable, **CVSROOT**, defines where `cvs` can find the repository. The user can place the software in any subdirectory of their area that they choose.

To retrieve a copy of the software, you need to go to the directory above where you want to install the software. `cvs` will make all the necessary subdirectories. For example, if you wanted to have the software installed in a directory called 'cvs' off of you home directory, the user would type:

```
cd
mkdir cvs
cd cvs
```

At this point the user can issue the cvs command by typing:

```
cvs checkout fnal
```

A copy of the software will be retrieved from the repository and placed in `$HOME/cvs/fnal`. The user can monitor the progress of the retrieval from the screen from which they issued the command.

The use of the software depends on having another Unix environment variable defined. For users of `csh` or its variants this definition should be entered in their `.login` file or put in a file and executed each time before using the software. Using the above directory structure, the user would set the `FNALROOT` variable as:

```
setenv FNALROOT $HOME/cvs/fnal
```

The software makefiles use this environment variable to know where to construct libraries as well as where to find include files.

Once a copy of the software is retrieved, the user must setup some directories that will be needed. Fortunately the added directories are created for the user in the Makefile. By issuing `make setup`, the user will create a library directory tree where teh various compiled libraries will go. There will be subdirectories under `$FNALROOT/lib` that correspond to which compiler and platform were used for the library build: `$FNALROOT/lib/gcc`, `$FNALROOT/lib/sun`, and `$FNALROOT/lib/sgi`.

Once the library tree is made, the user is ready to compile the libraries. By just typing `make`, the user will get a list of the options that can be used for

making the libraries. Depending on what platform the user is compiling on will dictate what options to use. For most users, using `make solaris-debug` will suffice. This will compile the libraries using the Solaris C++ compiler with debugging flags so that the code can be stepped through using the Solaris debugger. The libraries can also be compiled using the GNU g++ compiler, although this may not be supported in the future.

It usually takes a couple of hours to compile all of the libraries. Once the compilation is complete, the user can begin linking their own C++ programs to the libraries.

Chapter 3

Model Class layout

Presently the source code is broken down into 18 collections of classes. Each collection contains classes that provide a particular functionality. Below is a summary of the various collections and their contents.

beamline This contains the classes for constructing and performing various operations on a beamline. Included in this collection are the classes: `beamline`, `bmlnElmnt`, and `Proton`.

mxyzptlk This contains the Differential Algebra and Automatic Differentiation classes written primarily by Leo Michelotti. Included in this collection are the classes: `Jet`, `Map` and `LieOperator`.

basic_toolkit This contains general utility classes such as singly-linked lists (`slist`) and doubly-linked lists (`dlist`), as well as a `Complex`, `Vector`, and `Matrix` classes.

physics_toolkit This contains classes that perform a specific type of calculation. Presently the directory contains classes for calculating the fixed point for an orbit (`FPSolver`), as well as a class for the calculation of Edwards-Teng lattice functions (`EdwardsTeng`).

machines This directory contains subdirectories relating to model of various accelerators.

machines/Machine This directory contains the base class structure for the description of an accelerator machine. It contains a basis set of methods

one could use in the interrogation and manipulation of a model of a real accelerator.

machines/tev This collection contains the classes modeling the Tevatron both in Collider mode as well as in Fixed Target mode.

machines/swyd This collection contains the classes describing the different beamlines in the area of Switchyard user the former Accelerator Division jurisdiction.

machines/recycler This collection contains the classes describing the Recycler accelerator.

machines/mi_8gev This collection contains the classes describing the Main Injector 8 GeV line.

machines/mi This collection contains the classes describing the Main Injector.

machines/mr_8gev This collection contains the classes describing the obsolete Main Ring 8 GeV line.

machines/accumulator This collection contains the classes describing the Anti-proton source Accumulator storage ring.

server This collection contains the basic building blocks used in the construction of client/server applications. It consists of code to monitor a particular socket and to interpret the messages received there. A child process will be started at the next available socket to handle requests of a particular user.

tcl This collection provides a way to implement a Tcl/Tk interface on top of a C++ application by using a `Tcl_Object` as a base class.

sybase This collection provides classes for establishing connections to and retrieving data from Sybase database tables.

socket This collection is the socket++ class library which can be used to creating communication connections via sockets, pipes, and IP as well as classes for forking processes.

filter This is a directory containing various classes and routines that can be used to affect the reading in or output of beamlines. This directory contains, for example, routines for converting MAD input decks into files the class libraries can read.

3.1 Directory layout

To help the user better navigate through the maze of directories, this section will give you the general layout of the directory structure of the class library source code.

When the software classes are compiled, they produce libraries which can be linked into the user's program. These libraries reside in one of the subdirectories of the **lib** directory in the $FNALROOT directory depending on the platform and compiler used to compile them. There are three subdirectories under the **lib** directory; **lib/gcc**, **lib/sun** and **lib/sgi**. This **lib** directory tree is automatically constructed when issuing a `make setup` from the $FNALROOT directory.

Each of the major class structures above contains a directory structure that contains a **src**, **include**, and possibly an **app** and **lattices** directory.

The **src** directory contains the source code for the particular classes that make up that structure. The **include** directory contains the class declarations for the classes for that structure. The **app** directory contains application programs that primarily use or test the classes in the **src** directory, although they may also require other libraries.

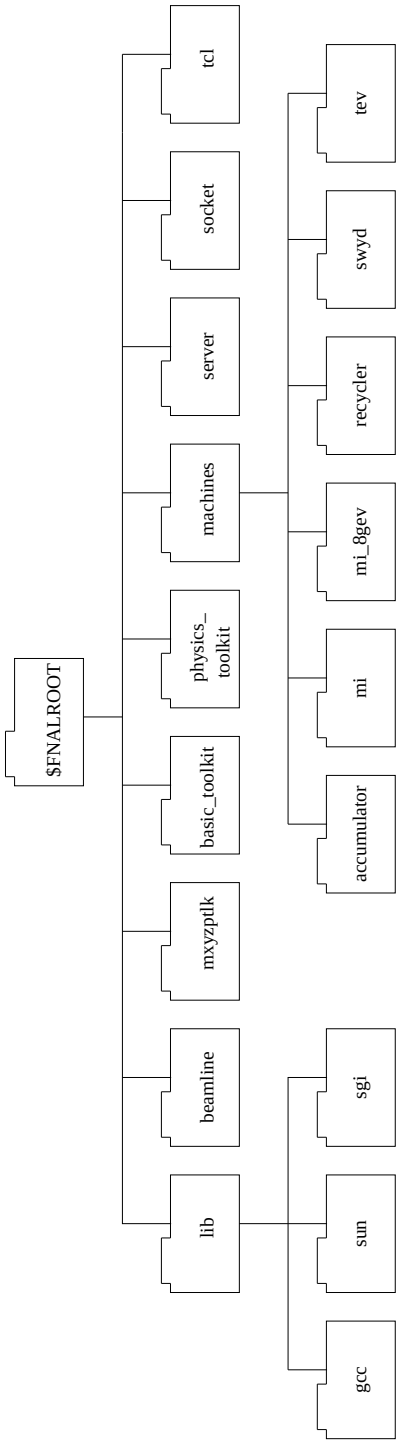

Figure 3.1: FNALROOT directory structure.

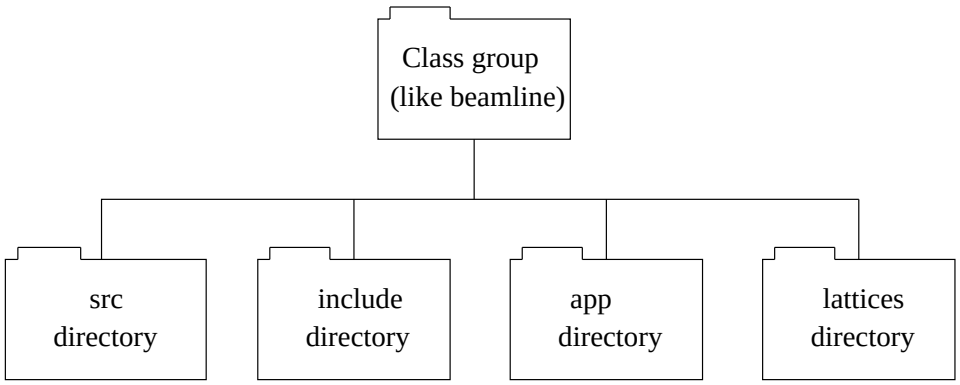

Figure 3.2: Class structure directory layout.

Chapter 4

Model Class Description

This chapter will provide a little more information as to the specifics of some of the more commonly used classes in each of the directories to help the user get better acquainted with the class hierarchies.

It will be helpful to the user to note that the units used for all beamline measurements and particle parameters in the classes are in MKS units.

measurement type	units	examples
transverse	meters	
longitudinal	meters	
angular	radians	
corrector strength	radians	
dipole strength	Tesla	
quadrupole strength	Tesla/Meter	

A valuable source of information as to the specific methods available for a particular class is the header files. The following descriptions of the classes will not cover the use of every method available, but will touch on some of the more common ones. Although the documentation in the files can be sparse at times, most of the method names are self-explanatory.

Chapter 5

Beamline classes

As mentioned in the introduction, the classes contained in this directory are used for the construction and manipulation of the beamlines. The basic component of a beamline in all accelerators is a beamline element as so it is with the classes in this directory. There is a base class called `bmlnElmnt` which all other classes inherit from that contains basic information about a beamline element such as its name, length, strength, alignment and methods to access and modify this information. All other beamline element-type classes (`hkick`, `vkick`, `drift`, etc.) inherit from this base class and extend this class as necessary.

The user can make instances of various `bmlnElmnt` classes but until they are tied together, they are just disconnected objects. As in a real machine where beamline elements are put together to make a beamline, so `bmlnElmnt`s are put together to make a `beamline`. A `beamline` is a class that inherits from both `bmlnElmnt` as well as a class not in this directory called `dlist` which is a doubly-linked list. The `beamline` class inherits from `bmlnElmnt` because you can construct a complex beamline element using `bmlnElmnt` components and group them together into a `beamline` and add it to another beamline, treating it as a beamline element.

The `beamline` class also inherits from `dlist` because a real machine beamline is really a 'list' of beamline elements where for each element, it is clear what the previous and next elements are. To add elements to a beamline, it is necessary to make an instance of the object to be added and then one can use the `beamline::append()` method to add the element to the beamline.

As the beamline is just a dlist of elements, one can access sequential

elements off of the list by using the `dlist_iterator`. This class has an overloaded method `dlist_iterator::operator()()` that returns a `void*` pointer to the object on the `dlist`. In the code example below, the above overloaded operator is used in the `while` loop to point to the next element on the `dlist` and assign it to a `bmlnElmnt*`. It is the responsibility of the user to cast this `void *` pointer to class type originally put on the `dlist`. Since all of the beamline elements inherit from `bmlnElmnt`, they all have the methods of this class defined. Casting to this base class allows access to the base methods for that class which is usually sufficient for most purposes. For instance, the user can enquire of type of the object using the `bmlnElmnt::Type()` method and based on the 'type' of this beamline element elect to take specific action. An example would be finding out if the `bmlnElmnt` presently pointed to was of type 'hmonitor' (a horizontal Beam Position Monitor). If so, the user could then print out the name of the horizontal BPM. The 'type' information can also be used to make an 'informed cast' to a more appropriate beamline element type. For instance, if the `type()` method returned a 'sbend' type, then the user could cast the base class `bmlnElmnt*` to the derived class `sbend*` and have access to the additional methods that an `sbend` element has.

As an example, below is a program that constructs individual beamline elements as well as a beamline and appends the elements to the beamline. Once the beamline is constructed, a `dlist_iterator` used used to go down the beamline and access each element of the beamline and print the name of the element.

```
#include "beamline.rsc"
#include <math.h>
#include <string.h>

int main(int argc, char* argv[]) {

  double ENERGY = 1000.00;

  beamline test;

  // define the elements in the beamline using a Tevatron.
  drift D1("D1", 6.12 );
  drift D2("D2", 6.12 );
```

```
  quadrupole TQF("T:QF", 0.762, -1.54420767985797624, 1);
  quadrupole TQD("T:QD", 0.762, -0.26895590398428737, 1);

  // Actually put the beamline together and set the energy.
  test.append(D1);
  test.append(TQF);
  test.append(D2);
  test.append(TQD);

  beamline* flatBeamline = test.flatten();
  beamline* lastBeamline = (beamline *)flatBeamline->Clone();

  lastBeamline->setEnergy(ENERGY);

  // Now setup to go down the beamline and access each
  // element to inquire about its name.
  dlist_iterator getNextElement(*(dlist*)lastBeamline);
  int counter = 1;
  while( (element = (bmlnElmnt *)getNextElement())) {
    cout << "Element #" << counter << " is a "
         << element->Name() << endl;
    counter++;
  }
}
```

As the pointer to each beamline element is retrieved from the beamline, the user can access the `bmlnElmnt` methods to determine what the name or type of the element and then take appropriate actions. The above example prints the name of each element retrieved from the beamline.

One of the uses of a beamline is for tracking particles through the beamline. To that end, there are a number of classes that inherit from a base class called `Particle`. This class allows the user to specify a particle with desired mass, energy, and charge and then propagate the particle down the beamline. Two of the more commonly used inherited classes from the `Particle` class are the `Proton` and the `Electron` class. These classes could be used to track a particle down the beamline and read off it's position at all of the hmonitors.

```
#include "beamline.rsc"
```

```
#include <math.h>
#include <string.h>

int main(int argc, char* argv[]) {

  double ENERGY = 1000.00;

  beamline test;

  // define the elements in the beamline using a Tevatron.
  drift D1("D1", 6.12 );
  hmonitor H1("H1");
  drift D2("D2", 6.12 );
  hmonitor H2("H2");
  quadrupole TQF("T:QF", 0.762, -1.54420767985797624, 1);
  hmonitor H3("H3");
  quadrupole TQD("T:QD", 0.762, -0.26895590398428737, 1);
  hmonitor H4("H4");

  // Actually put the beamline together and set the energy.
  test.append(D1);
  test.append(TQF);
  test.append(D2);
  test.append(TQD);

  beamline* flatBeamline = test.flatten();
  beamline* lastBeamline = (beamline *)flatBeamline->Clone();

  lastBeamline->setEnergy(ENERGY);

  // Use a single Proton for tracking.
  Proton p;
  // Coordinate order: x    y   cdt   x'   y'  dp/p
  double coords[6] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
  p.energy(ENERGY);
  p.setState(coords);

  // Setup to go down the beamline looking for any elements
```

```
  // that are of the type 'hmonitor'.
  dlist_iterator getNextElement(*(dlist*)lastBeamline);
  int counter = 1;
  while( (element = (bmlnElmnt *)getNextElement())) {
    element->propagate(p);
    if(!strncmp(element->Type,"hmonitor",strlen("hmonitor")) {
      p.getState(coords);
      cout << "At " << element->Name()
           << " the coordinates are:\n   ";
      for(int i = 0; i<6; i++){
        cout << coords[i] << "  ";
      }
      cout << endl;
    }
  }
}
```

In the above code, the six coordinates of the `Proton` that are retrieved at each `hmonitor` are in the following order: x (in meters), y (in meters), $c\Delta t$ (in meters), x' (in radians), y' (in radians), $\frac{dp}{p}$ (unitless).

If the user wanted to use a distribution of particles, there's a class for that too. The class is called `ParticleBunch` class. It inherits from slist and can make a distribution of particles using whatever distribution function the user provides, conforming to distribution sigmas and offsets provided by the user. There is a `Gaussian` class that can be used in conjunction with the `ParticleBunch` class to make a particle bunch with a Gaussian distribution in all six cooridinates. By modifying the above example, we can use the code to propagate a ProtonBunch rather than a Proton.

```
  // Use a ProtonBunch for tracking.
  ProtonBunch pb;
  // Beam sigma order is the same a the coordinate order:
  //                   x    y   cdt   x'   y'  dp/p
  double sigmas[6] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
  // Need a distribution for the particles.
  // Let's use a Gaussian distribution.
  Gaussian dist(0);
```

```
pb.recreate(1000,ENERGY,sigmas,&dist);

dlist_iterator getNextElement(*(dlist*)lastBeamline);
slist_iterator getNextProton((slist )pb);
int counter = 1;
while( (element = (bmlnElmnt *)getNextElement())) {
  element->propagate(pb);
  if(!strncmp(element->Type,"hmonitor",strlen("hmonitor")) {
    Proton* p;
    while( p = (Proton *)getNextProton()){
      p->getState(coords);
      cout << "At " << element->Name()
           << " the coordinates are:\n    ";
      for(int i = 0; i<6; i++){
cout << coords[i] << "  ";
      }
      cout << endl;
    }
  }
}
```

Another common action to perform on a beamline is to calculate the Twiss parameters. Depending on whether the user wants the beamline to represent a ring or a single-pass beamline will determine what parameters are passed to the `Twiss()` method. For a single-pass beamline it is necessary to pass to the `Twiss()` method the starting Twiss parameters. Fortunately there is a struct designed to hold the Twiss parameters called `lattFunc` which is defined as follows:

```
struct lattFunc : public BarnacleData {
  double arcLength;
  struct {
    double hor;
    double ver;
  } dispersion;
  struct {
    double hor;
    double ver;
```

```
  } dPrime;
  struct {
    double hor;
    double ver;
  } beta;
  struct {
    double hor;
    double ver;
  } alpha;
  struct {
    double hor;
    double ver;
  } psi;

  lattFunc();
  ~lattFunc() {}
  lattFunc& operator=( const lattFunc& );
};
```

For a single-pass beamline, this struct should be set to the initial Twiss parameters at the beginning of the beamline and passed to the `Twiss()` method.

For a beamline representing a ring, the Twiss parameters must match the boundary conditions of being continuous at the ends of the beamline, therefore it is not necessary to provide initial Twiss parameters.

In either case, the Twiss method is called and the Twiss parameters are calculated for each beamline element in the beamline. The Twiss parameters are not returned to the calling routine, but are 'attached' to each of the beamline elements using a class appropriately called a `Barnacle` class. To retrieve the Twiss for all or some of the beamline elements in the beamline, the user would go down the beamline using the `dlist_iterator` and at the beamline elements of interest, use one of the `beamline::whatIsLattice` methods to retrieve the attached Twiss parameters.

Chapter 6

Mxyzptlk classes

This collection of classes contains a package of classes for using Differential Algebra techniques as well as utility classes. Examples of the classes that fall into the Differential Algebra category are `Jet`, `Mapping`, and `CLieOperator`.

Leo Michelotti has a very good write-up on the use of the Differential Algebra classes which the reader is referred to.

Chapter 7

Machine classes

This collection of classes contains descriptions of the various accelerators used at Fermilab. Under the `machines` directory are the individual machine models along with the base class `Machine` that they all inherit from.

The basic inheritance structure is as follows:

In this diagram, the myMachine is a hypothetical machine. The `Machine` class which itself inherits from the `beamline` class provides basic methods to access characteristics of a generic accelerator machine. Taking a look at the `Machine` header file below will give the reader an idea of what methods are available.

```
class Machine : public beamline {
protected:
  double energy; // The energy is common to all classes
// This class should not be constructed
// on its own.
  Machine();
public:
  virtual ~Machine();

//------------------ Public Variables ----------------------
// The design is cleaner if these variables are made public

  dlist circuitList; // This is a list of circuits for
// all powered elements of the lattice.
  int numCircuits; // How many circuits
```

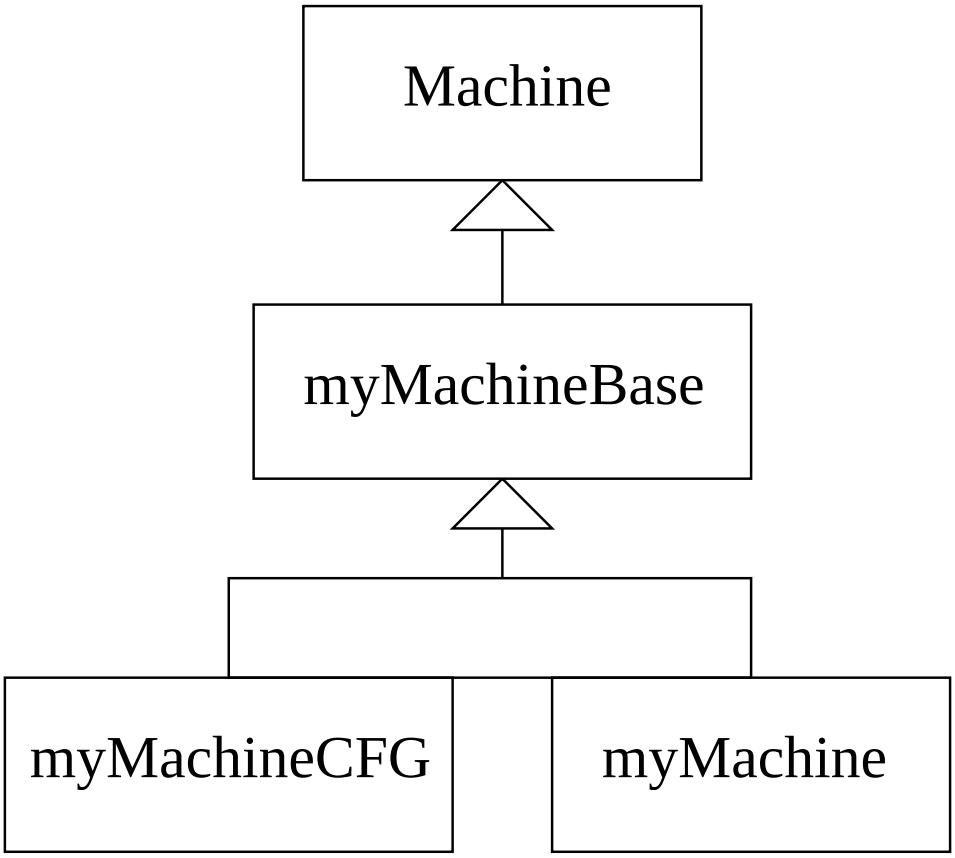

Figure 7.1: Machine Inheritance tree.

```
  dlist moverList; // This is a list of movers for
// all powered elements of the lattice.
  int numMovers; // How many movers

//----------------- Methods --------------------------------

  void setAlign(char*,// element name
alignmentData); // alignment data
  void set(char*,// element name
   double); // strength value;
  virtual void readConfig(char*);  // strength table file name
```

```
  virtual void readLattice(char*); // read in lattice from file
  virtual void buildCircuits();
  virtual void buildMovers();
};
```

As we can see from the above header file, there are various public data members that contain information about how many circuits are in the machine, as a `dlist` of the circuits in the machine. Each magnet can also be moved, and hence, there is a `dlist` of magnet movers. Both of these lists can be created by the user using methods provided by the class.

Moving onto the class methods, one can see that the user is given methods to change the strength or the alignment of an element if the user knows the name of the element. As promised, there are also methods for building the lists of circuits and movers.

The hypothetical `myMachineBase` class is abstract and directly inherits from the `Machine` class. Its purpose is to provide some specialization of the basic `Machine` class for a particular type of machine. For instance, the `ColliderBase` class declared in `$FNALROOT/machines/tev/include` provides additional methods specific to the Tevatron Collider model. These include methods such as `ColliderBase::buildFeeddownCircuits()`, `ColliderBase::separatorOn` and
`ColliderBase::pbarFixedPoint()`.

Under this class are two other inherited classes called `myMachineCFG` and `myMachine`. The difference in these two classes is the way that the definition of the physical tunnel elements in beamline or machine are defined. In the `myMachineCFG` class, the class constructor includes a C++ file that describes the beamline in terms of the `beamline` class library. Therefore, the defintion of the particular machine is fully defined at library compile time. The definitions or ".cfg" files are usually located in the particular machines include directory such as `$FNALROOT/machines/tev/include`.

For the Tevatron in collider mode, the ".cfg" file defines the basic lattice of the machine at 1 TeV. To get the changes to the lattice for various low beta squeeze values, a file is read in to describe that particular low beta step. For instance, if the user wanted the Collider model representing the lattice for BD step 1, the user would make a `ColliderCFG` model and pass it the name of a file describing the particular elements changed for BD step 1. This normally consists of the low beta magnets, tune circuits, separators and feeddowns. These modifications to the lattice are kept in ".dat" files in

the `lattices` directory.

The second inherited class from `myMachineBase` is the `myMachine` class. This class provides an ASCII file describing the physical tunnel elements in the beamline or machine to be specified at runtime. This class passes the file read in to a stream parser in the `beamline` class libraries which then instantiates the various beamline elements as the file is read in. If the user wishes to edit a particular element, this runtime machine definition scheme provides an easy way to accomplish this. The files used for initialization are usually found in hte machines lattice directory such as `$FNALROOT/machines/tev/lattices` and have a suffix of ".bml".

Below will be given an example of each type of accelerator machine instantiation so that the user can get a feel for how to use these pre-defined accelerator models. The first is a modification of the `twissCheck.cc` application in the Tevatron 'app' doirectory. This uses the "CFG" format file for hte beamline initialization.

```
#include <math.h>
#include <string.h>
#include <iomanip.h>
#include "beamline.rsc"
#include "ColliderCFG.h"

int main( int argc, char** argv ) {

  // Create the Jet enviroonment
  Jet::BeginEnvironment( 1 );
  coord x(0.0),  y(0.0),  z(0.0),
  px(0.0), py(0.0), pz(0.0);
  Jet__environment* pje = Jet::EndEnvironment();
  JetC::lastEnv = JetC::CreateEnvFrom( pje );
  JetC__environment* pjeC = JetC::lastEnv;

 // Construct the model ring
  int i;
  ColliderCFG* tev;

  // Read in the changes from Collider at 1 TeV vs. the
```

```
  // Collider at BD step 1.
  tev = new ColliderCFG("../lattices/h1000s1.dat");
  tev->setEnergy(1000.00044018);

  lattFunc W;
  lattRing initR;
  double energy = tev->Energy();
  JetProton p(energy);
  // Calculate the Twiss parameters treating the Collider
  // as a ring.
  int result =  tev->twiss(p);

  // Retrieve the Tunes.
  initR = tev->whatIsRing();
  cout << "Tunes = " << initR << endl;

  // Now got through the machine and print out the
  // Twiss parameters at every beamline element.
  i = 0;
  bmlnElmnt* element;
  dlist_iterator getNext(*(dlist*)tev);
  while( (element = (bmlnElmnt *)getNext())) {
    W = tev->whatIsLattice(i);
    i++;
    cout << element->Name()    << " " << W.arcLength << " "
         << W.alpha.hor        << " " << W.beta.hor  << "  "
         << W.psi.hor/M_PI/2.0 << " " << W.alpha.ver << " "
         << W.beta.ver         << " " << W.psi.ver/M_PI/2.0
         << endl;
  }
}
```

The second example is the same as the above example, but the beamline initialization will be done using a ".bml" file.

```
#include <math.h>
```

```
#include <string.h>
#include <iomanip.h>
#include "beamline.rsc"
#include "Collider.h"

int main( int argc, char** argv ) {

  // Create the Jet enviroonment
  Jet::BeginEnvironment( 1 );
  coord x(0.0),  y(0.0),  z(0.0),
  px(0.0), py(0.0), pz(0.0);
  Jet__environment* pje = Jet::EndEnvironment();
  JetC::lastEnv = JetC::CreateEnvFrom( pje );
  JetC__environment* pjeC = JetC::lastEnv;

 // Construct the model ring
  int i;
  Collider* tev;

  // Read in the particular lattice (e.g. helix step 1)
  tev = new Collider("../lattices/h1000s1.bml");
  tev->setEnergy(1000.00044018);

  lattFunc W;
  lattRing initR;
  double energy = tev->Energy();
  JetProton p(energy);
  // Calculate the Twiss parameters treating the Collider
  // as a ring.
  int result =  tev->twiss(p);

  // Retrieve the Tunes.
  initR = tev->whatIsRing();
  cout << "Tunes = " << initR << endl;

  // Now got through the machine and print out the
  // Twiss parameters at every beamline element.
  i = 0;
```

```
  bmlnElmnt* element;
  dlist_iterator getNext(*(dlist*)tev);
  while( (element = (bmlnElmnt *)getNext())) {
    W = tev->whatIsLattice(i);
    i++;
    cout << element->Name()    << " " << W.arcLength << " "
         << W.alpha.hor        << " " << W.beta.hor  << "  "
         << W.psi.hor/M_PI/2.0 << " " << W.alpha.ver << " "
         << W.beta.ver         << " " << W.psi.ver/M_PI/2.0
         << endl;
  }
}
```

7.1 Tevatron classes

The `machines/tev` directory contains classes that are specific to the two modes that the Tevatron machine can run in: Collider and Fixed Target. For Collider mode, there are the `Collider` family of classes. The Collider models are set for the design energy of 1 TeV momentum.

For Fixed Target mode, there are the `TevExtract` family of classes. These classes can combined with the classes in the `machines/swyd` directory to make a Tevatron that can extract to one or many beamlines. The Fixed Target model is setup to model the Tevatron at 800 GeV during fast spill (as opposed to slow spill).

The directory also contains support classes for Luminosity and RF bucket cogging calculations.

7.2 Switchyard classes

The `machines/swyd` directory contains classes that describe the various beamlines that are run in Fixed Target mode as of 6/97. This includes the Proton, Meson, Neutrino and Muon beamlines. These beamlines can be attached to the fixed Target model of the Tevatron (`TevExtract`). These beamlines contain electrostatic septa and three-way lambertson magnets. The electrostatic septa are modeled as a zero-length kick element with drift elements making

up the length of the septa magnet. The lambertsons are modeled by having a zero-length `thinLamb` beamline element that defines the locations of the septa. As a particle bunch intercepts this element, the element sorts the particles based on whether the particles are transversely higher or lower than the septa, removing particles from the parent bunch and placing them in a new particle bunch.. The `thinLamb` beamline element has attached to it a new beamline representing the extraction beamline. The particles that pass below the septa continue to the next beamline element in the beamline. The particles that pass above the magnet septa are passed to a new beamline and a new propagation is begun.

7.3 Recycler classes

The `machines/recycler` directory coontains classes that describe the Recycler storage ring. The two classes of interest are the `Recycler` and the `RecyclerCFG` class - with the differences being explained previously.

7.4 Main Injector 8 Gev Line classes

The `machines/mi_8gev` directory contains the classes describing the Main Injector 8 GeV line that goes from Long-3 in Booster to the Main Injector accelerator. This directory contains the two classes `MI_8gev` and `MI_8gevCFG`. Since the '.cfg' file is generated from a MAD input deck, the names in the '.cfg' file are not necessarily very descriptive names. The '.bml' file (to be used with the `MI_8gev` class contains ACNET names for the powered beamline elements as well as the latest values for magnet strengths as well as SWICS (Segmented Wire Ionization Chamber) represented by `mwireMonitor` beamline elements.

7.5 Main Injector classes

The `machines/mi` directory contains th classes that describe the Main Injector at 8 GeV.

Chapter 8

Basic toolkit classes

This library contains many general purpose classes that can be used in a wide variety of applications beyond Accelerator modeling. These classes include `dlist`, `VectorD`, `Matrix`, and `IntArray`. Examples of some of these classes will be given below.

The `dlist` class provides a doubly-linked list class that can be used to store, add and delete any number of like-objects. A `dlist` works like a variable size array of like-objects. They are useful if you do not know how many objects you are going to have to store beforehand.

The example below uses a simple struct of a name and an age as the 'form' of the type of objects to be stored.

```
#include "dlist.h"
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <iomanip.h>

//*************************************************************
//
//  This is a test program the uses dlists.  It reads in 5
//  names and then stores them on the list for later retrieval.
//
//*************************************************************
```

```
struct dlist_entry_type {
  char* name;
  int   age;
};

int main(void){

  dlist NameList;         // This defines the dlist object.

  cout << "This program will prompt the user for 5 names "
       << "and ages.  The\nnames will be put on a dlist "
       << "and read back to the user." << endl;

  char tmpBuffer[80];          // Temporary storage for the
                               // user input.
  dlist_entry_type* newData;   // This is what will appear
                               // on the list.
  for(int i = 0; i<5; i++){
    newData = new dlist_entry_type;  // Allocate the struct.
    cout << "enter name:";
    cin.getline(tmpBuffer,80,'\n');

    // Now that we have the name, make room for it in
    // the struct.
    newData->name = new char[strlen(tmpBuffer)+1];
    strcpy(newData->name,tmpBuffer); // Copy name to struct.
    cout << "enter age:";
    cin.getline(tmpBuffer,80,'\n');
    newData->age = atoi(tmpBuffer);  // Convert age to integer.

    // Now that we have the struct filled out with the
    // name and age, put it on the list.
    NameList.append(newData);
  }

  cout << "The next name/age combination will be inserted "
       << "between the second and third entries made above"
       << endl;
```

```
  newData = new dlist_entry_type;  // Allocate the struct.
  cout << "enter name:";
  cin.getline(tmpBuffer,80,'\n');

  // Now that we have the name, make room for it in the struct.
  newData->name = new char[strlen(tmpBuffer)+1];
  strcpy(newData->name,tmpBuffer);   // Copy name to struct.
  cout << "enter age:";
  cin.getline(tmpBuffer,80,'\n');
  newData->age = atoi(tmpBuffer);   // Convert name to integer.

  // Now that we have the struct filled out with the name
  // and age, put it on the list.
  NameList.putBelow(NameList[3], newData);


  cout << "\n\nNow that the names have been read in, they "
       << "will be output one at a time. " << endl;

  dlist_iterator     getNext(NameList);  // This is used to
                                         // go down the list.
  dlist_entry_type*  tmpData;

  // Pull an element off the list and type-cast it to the
  // correct type of pointer.
  while(tmpData = (dlist_entry_type *)getNext()) {
    cout << "Name:  " << tmpData->name
         << " age: " << tmpData->age << endl;
  }
}
```

The code places 5 names and ages on the dlist and then inserts a sixth name between the second and third object on the dlist. Finally, a dlist-iterator is used to go down the dlist and retrieve the pointers to every object on the list, in this case displaying the data members of the structure.

It is important to note that the `dlist` is just a collection of objects and that it provides no method for retrieving the elements off the list. This is

left to another class called a `dlist_iterator`. This design allows multiple accesses to be made by `dlist_iterators` to the same dlist and the same time as would be done if the elements on the `dlist` were being sorted. Also note that the `dlist` is really just a list of memory addresses and as such, when they are retrieved off of the `dlist`, they must be cast to the same type or a base class type that they represent. There are other types of iterators that can be used for `dlists` that specific characteristics for specific applications.

There is also a `slist` or singly-linked class as well which works pretty much the same as the `dlist` with the following exception: in a `dlist`, you have access to the elements on the `dlist` both above and below the present element. In an `slist`, you have access only to the element below the present element. so for example, if you have the address of the last element on a `dlist`, you can travel back up the list to the top whereas for an `slist` you can not.

Another useful class is the `MatrixD` class. There are variants of this class based on the type of elements in each 'cell' being, integer, double or complex. In the following discussion, the `MatrixD` class will be discussed. The major advantage of the use of the `MatrixD` class is that the computations normally done with matrices can be written down in a compact easy-to-read form rather than the more obscure nested 'for' loop format.

The `Vector` class is another useful class for general mathematical calculation. The `Vector` can be constructed with any number of elements, but default constructor will result in a three element zero `Vector`. The example below is one from Leo Michelotti that demonstrates some of the uses of the `Vector` class.

```
/*
 * VectTest.cc
 *
 * Demonstration.
 * Leo Michelotti  March 22, 1995
 *
 */

#include "Vector.h"

main()
```

```
{
  Vector a(4), b(4), c(4);
  for( int i = 0; i < 4; i++ ) {
    a(i) = i;
    b(i) = 2.0*i;
  }

  if( a.IsNull() ) cout << "Wrong a\n";
  else             cout << "Right a\n";

  if( b.IsUnit() ) cout << "Wrong b\n";
  else             cout << "Right b\n";

  c = 3.0*a + b;

  cout << "a =" << a        << endl;
  cout << "b =" << b        << endl;
  cout << "c =" << c        << endl;
  cout << ( 4.0*c - a + b ) << endl;
}
```

Chapter 9

Physics toolkit classes

This collection of classes provide general accelerator modeling classes that are not limited to one particular accelerator model. Presently, the two classes provided are the `EdwardsTeng` class for calculating beamline parameters (similar to Twiss) for a coupled lattice and the `FPSolver` class for calculating the orbital fixedpoint for a ring-like accelerator.

Chapter 10

Server classes

The server collection of classes comprise the foundation for server-side of the client/server online model and Open Access Server both which are accessible through the Accelerator controls system console applications pages. The Online Models are applications that reside on a single PA. The user can start the particular application and modify a model of the accelerator. This includes in many instances, reading real accelerator parameters and comparing them to the results of the model. This has the benefit of allowing the user to modify characteristics for the accelerator that may not correspond to an ACNET device (alignment of beamline elements, for instance).

The Open Access Model provides via a re-direction service the ability to have any console application retrieve data from a model of the accelerator. In this way, the application does not know it is re-directed and retrieves data not from the real accelerator front-end but from the Open Access Model. This provides a testing bed for applications written before accelerators have been commissioned. Once an accelerator is running, it allows comparision of model-generated data to be compared to real accelerator ACNET data.

The directory contains code for the base class `serveBase` that all the other model servers inherit from. The rest of the classes are inherited classes for specific machines. Presently, the modeled machines include Online Models for the Main Injector 8 GeV line, the Main Injector, the Recycler, the Collider, and the Tevatron and Switchyard in Fixed Target mode. For the Open Access Server, there exist BL8erve (the prototype OAM for the Main Ring 8 GeV line, now obsolete) and the Main Injector.

10.1 Anatomy of the Client/Server software

This section will attempt to summarize the client/server application environment used for the Online Model and to some extent the Open Access Client. There will be a description of the database table structure and usage as well as a description of the client/server handshaking process used for passing messages.

The server software is developed to make use of C++ accelerator modeling code developed primarily by Leo Michelotti and Jim Holt, the publicly available socket++ socket library and an in-house wrapper class on the Sybase Open Client libraries. Together, these make available various models of accelerators. The primary client-side software has been developed with the ACNET controls system in mind, but there is nothing inherent in the server side that would disallow another type of GUI application to communicate with the server. The only requirement would be that the GUI application would need access to the model database server.

The communication with the server is done through TCP/IP sockets through which ASCII messages are passed back and forth. Most of the messages are used as requests for calculations. Both the clients and the server have access to a database server which acts as an intermediary for the exchange of the actual calculation results. This database solution was found to be much faster than trying to stuff large structures of data through a TCP/IP socket. A diagram of this communcation scheme is shown in Figure 10.1.

To better visualize the interplay between the user, the masterServer and the accelerator model, a timeline of the model startup is given in Table 10.1.

At this point, the accelerator-specific model is now a separate process communicating with the user and satisfying requests.

There exists in the model database, tables representing twiss data, orbit data, lists of elements and circuits for each of the lattices of the various models. These tables are constructed from design parameters whenever the design model changes – which is typically not very often. These tables are used as the starting point for the model so that massive calculations of various accelerator parameters do not have to be performed at startup. When the machine-specific model is started, a copy of 'view' of the lattice tables is made for the user. In this way, the user is insulated from changes that other users could be making to another model of the same accelerator. This copy is maintained as long as the user needs the tables. Once the communication is

user	**masterServer**	**accelerator-specific model**
	masterServer listening to port 3000	non-existent
Model startup request.	Parses message to determine which model is requested and forks a process to start that particular model.	
		Figures out what database slot is open and uses that as an offset from port 3000 for it's communication. It informs masterServer of its port.
	Passes model port number on to user.	
User specifies to model what lattice to use from initialization.		
		Model initializes with user-specified lattice. This entails copying database lattice tables into a particular view for the user, so that in essence the user has his own model.

Table 10.1: Model timeline for startup.

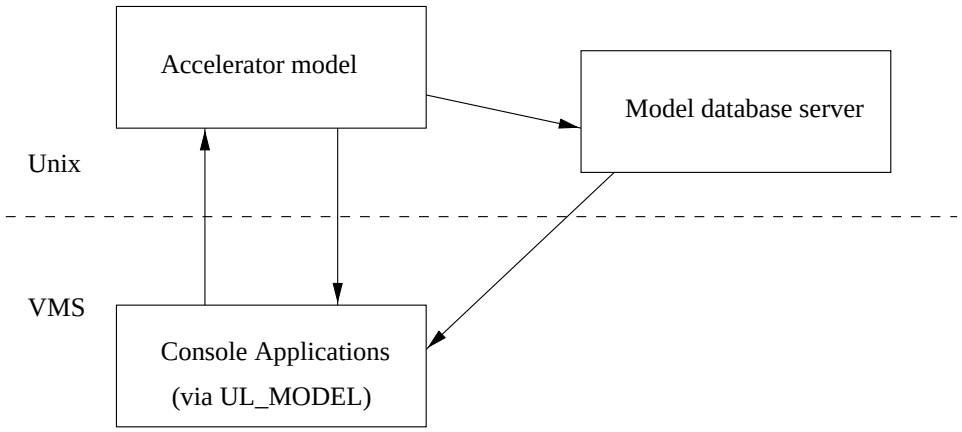

Figure 10.1: Client/Server/Database communication.

broken, the tables are truncated and the database slot is freed up for another user.

As long as there are no changes to the strengths of the model beamline elements, these tables represent the accelerator parameters for the particular lattice. Once a change is made to a magnetic element in the model, the 'true' accelerator parameters and those in the database tables are out of sync. When the user asks for Twiss parameters, after making a change, the model re-calculates new Twiss parameters and bulk copies them into the user's database view. Updates to the tables are not done unless the user a) changes a magnetic element in the model and b) requests accelerator data from the database tables. In this way, the model only refreshes data that the user is really interested in.

If the user changes one of the model beamline element strengths, then the values of the tables are marked as stale, and further requests for accelerator parameters will force a new calculation of those particular parameters requested. This means that if a quadrupole strength is changed, and then the user requests new Twiss parameters, the model will calculate new Twiss parameters, and copy those new values to the Twiss database table. Note that in this scenario, the orbit data, as an example, has not been re-calculated and as this scenario stands does not reflect the present state of the model. If the user were to ask for Orbit data now, the model would calculate new orbit data and copy it into the orbit database table.

Chapter 11

Tcl classes

The Tcl collection of classes provide the user with the means to make a program with a Graphical User Interface (GUI). The language Tcl/Tk is useful for making graphical interfaces with little work and the learning curve is not very steep.

The base class for this collection of classes, is the `Tcl_Object` class. This class can be used as the base class in a user application with the user providing methods to be invoked in response to pushbuttons or entry in data fields.

Below is part of the code for a very simple Rolodex based on using the `Tcl_Object` class. This example can be thought of as the GUI version of the `dlist` example on page 32

```
#ifndef TCLLIST_H
#define TCLLIST_H
#include "dlist.h"
#include "Tcl_Object.h"

struct dlist_entry_type {
  char* name;
  int   age;
};

class rolodex : public Tcl_Object {
 private:
  dlist*  NameList;
```

```
  char*  tclFileName;

 protected:
  void  fileName_to_tcl(char*);
  char* get_filename();
  void  exechook();
 public:
  rolodex();
  ~rolodex();

// member functions callable from the tcl/tk script.
  void save_entry();
};

#endif  // TCLLIST_H
```

Here is the implementation code for the above class.

```
#include "tclList.h"
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <iomanip.h>

//*******************************************************
//
//  This is a test program the uses dlists.  It reads in
//  names and ages and then stores them on the list for
//  later retrieval.
//
//*******************************************************

extern Tcl_CmdProc c_save_entry;

rolodex::rolodex() {
  init_interp();
```

```
  NameList = new dlist;

  CreateCommand ("c_save_entry", c_save_entry);
}

rolodex::~rolodex() {
;
}

void rolodex::exechook(){
//*****************************************
//
//  method to initialize other interpreters.
//
//*****************************************
;
}

char *rolodex::get_filename(){
//*****************************************
//
//  method to recall the tcl script to feed to wish
//
//*****************************************
  return("./myRolodex.tcl");
}

void rolodex::fileName_to_tcl(char* name) {
//*****************************************
//
//  method for converting filenames to somethign tcl
//  understands.
//
//*****************************************
  Tcl_SetStrVar("fileName", name);
}

void rolodex::save_entry() {
```

```
//*****************************************
//
//  method for saving an entry made onto the dlist.
//
//*****************************************
  int   NewAge;   // Not THE new age, it's just a
                  // variable name.

  // Retrieve the new name and age to be put on the
  // list.  All the Tcl_ methods are from Tcl_Object.

  char* NewName = Tcl_GetStrVar("name"); // NO NULL_TERMINATOR!
  NewAge = Tcl_GetIntVar("age");

  // Put the new name/age on the dlist.
  dlist_entry_type* NewEntry;
  NewEntry = new dlist_entry_type;
  NewEntry->name = new char[strlen(NewName)+1];

  strncpy(NewEntry->name,NewName,strlen(NewName));
  // We must NULL-terminate the string.
  NewEntry->name[strlen(NewName)] = '\0';
  NewEntry->age = NewAge;
  NameList->append(NewEntry);

  // Just for grins, lets print out what's in the
  // dlist at this point.

  dlist_iterator     getNext(*NameList);
  dlist_entry_type*  tmpData;

  // Pull an element off the list and type-cast.
  while(tmpData = (dlist_entry_type *)getNext()) {
    cout << "Name:  " << tmpData->name << " age: "
         << tmpData->age << endl;
  }
  return;
}
```

```
// These macro calls actually define the plain-C embedding:
// CAUTION: whitespace between parentheses can cause errors!
// These are where the connection is made to procedure calls
// in Tcl/Tk and the associated method in C++.

_CMD_Embedding(rolodex,c_save_entry,save_entry);
```

Now the above example has just a GUI on a basic type of program. If the user wants to put a GUI on a program to model an accelerator, then there is another class called `modelGUI` that provides a base class that the user can build upon. This approach is what is used by Jim Holt and Andrew Braun or their interactive model of the Final Focus Test Beam line at SLAC.

Chapter 12

Sybase classes

The classes contained in the Sybase directory encapsulate many of the common actions that the model has with the database. The `SybaseCom` class is basically a C++ wrapper on the Sybase client libraries. It provides atomic methods to perform querys and table insertions.

The `LatticeDB` class is designed for loading the model database with the lattice tables, which are the starting point for the Online Models. This class is usually used when a new lattice is being put into the database.

The `ModelDB` class provides communication between the model database tables and the model itself. In fact, the server classes in the `server` directory contain a pointer to a `ModelDB` object.

The `SlotDB` class is a small utility class that goes though the different views and looks for an open slot. It then reports back this slot number. This class is used during Online Model startup.

Chapter 13

Socket classes

The socket classes are those developed by Gnanasekaran Swaminathan and available in his socket++ package. These classes allow socket communication of various modes with little programming on the user's part. The reader is directed to his documentation for an excellent description of the features and abilities of this class library.

Chapter 14

Filter classes

The filter classes contain various useful classes and routines to perform manipulations on beamlines and such. There are also Perl scripts that can be used to generate beamline files in ".cfg" format from MAD.

The `concatDrifts` routine allows the user to pass a beamline into the `concatDrifts` method and have returned a beamline with all of the multiple drifts concatenated into one. This can be used to speed up tracking by reducing the total numbe rof elements to track through.

The `beamlineStreams` class provides a way to read and write a beamline to a file in ".bml" format.

The `writeMAD` routine provides a way to output a beamline in MAD format.

The `scaleBeamline` routine provides a way to scale the strength of beamline elements by a factor.

The `setBunches`, `ForwardBucket` and `ReverseBucket` classes provide ways to manipulate the distribution of particles in RF buckets. These classes can be used in modeling of various cogging scenarios.